Efficient Algorithms for the Tree Homeomorphism Problem

Michaela Götz[1], Christoph Koch[1], and Wim Martens[2*]

[1] Saarland University
Saarbrücken, Germany
{goetz,koch}@infosys.uni-sb.de
[2] University of Dortmund
Dortmund, Germany
wim.martens@udo.edu

Abstract. Tree pattern matching is a fundamental problem that has a wide range of applications in Web data management, XML processing, and selective data dissemination. In this paper we develop efficient algorithms for the tree homeomorphism problem, i.e., the problem of matching a tree pattern with exclusively transitive (descendant) edges. We first prove that deciding whether there is a tree homeomorphism is LOGSPACE-complete, improving on the current LOGCFL upper bound. As our main result we develop a practical algorithm for the tree homeomorphism decision problem that is both space- and time efficient. The algorithm is in LOGDCFL and space consumption is strongly bounded, while the running time is linear in the size of the data tree. This algorithm immediately generalizes to the problem of matching the tree pattern against all subtrees of the data tree, preserving the mentioned efficiency properties.

1 Introduction

Tree patterns are a simple query language for tree-structured data. They are at the heart of several widely-used Web languages such as XPath and XQuery [4]. As a consequence, they form part of a number of typing mechanisms such as XML Schema, and of Web Programming Languages. They have also been used as query languages in their own right, for example for expressing subscriptions in publish-subscribe systems [1, 5, 6, 13].

The general tree pattern matching problem considered in the literature is the problem of finding a mapping between two node-labeled trees which is, in a sense, a cross of a subtree homomorphism and a homeomorphism. In this paper we consider a clean and important special case of the tree pattern embedding problem that we call the *tree homeomorphism problem.* The question we consider is whether there is a mapping θ from the nodes of the first tree, the *tree pattern*

* This work was supported by a scholarship of the FWO-Vlaanderen that permitted Wim Martens to visit the Technical University of Vienna in January–February, 2005.

	time	space	streaming
Yannakakis 1981 [19]	$O(\|Q\| \cdot \|D\| \cdot \mathit{depth}(D))$	$O(\mathit{depth}(Q) \cdot \|D\|)$	no
Gottlob et al. 2002 [10]	$O(\|Q\| \cdot \|D\|)$	$O(\|Q\| \cdot \|D\|)$	no
Olteanu et al. 2004 [16]	$O(\|Q\| \cdot \|D\| \cdot \mathit{depth}(D))$	$O(\|Q\| \cdot \mathit{depth}(D) + \|D\|)$	yes
Bar-Yossef et al. 2005 [3]	$O(\|Q\| \cdot \|D\|)$	$O(\|Q\| \cdot \log \|D\| + \mathit{cand}_D)$	yes
Ramanan 2005 [17]	$O((\|Q\| + \mathit{depth}(D)) \cdot \|D\|)$	$O(\|Q\| \cdot \mathit{depth}(D) + \mathit{cand}_D)$	yes
Our bottom-up algorithm	$O(\|Q\| \cdot \|D\| \cdot \mathit{depth}(\|Q\|))$	$O(\mathit{depth}(D) \cdot \mathit{branch}(D))$	no
Our LOGSPACE algorithm	$\mathit{poly}(\|Q\| + \|D\|)$	$O(\log(\|Q\| + \|D\|))$	no

Table 1. Time and space consumption for algorithms solving the tree homeomorphism matching problem. Here $\mathit{depth}(\cdot)$ and $\mathit{branch}(\cdot)$ denote the depth and maximal branching factor of a tree, respectively.

or *query*, to the nodes of the second tree, the *data tree*, such that if node y is a child of x in the first tree, then $\theta(y)$ is a *descendant* of $\theta(x)$ in the second tree. We also consider the *tree homeomorphism matching problem*: finding *all* nodes v of the data tree such that there is such a tree homeomorphism with v the image of the root node of the pattern tree. This problem of selecting all nodes whose subtrees match the tree pattern has frequent application in XML and Web query processing [1, 10].

While this problem is of immediate practical relevance and a substantial number of papers have studied complexity and efficient algorithms for tree pattern matching, the precise complexity of both the general tree pattern matching problem and the tree homeomorphism problem are open; they are both known to be in LOGCFL and LOGSPACE-hard [11].The former can be immediately concluded from earlier results on the complexity of the acyclic conjunctive queries [12] and the positive navigational fragment of XPath [11], both much stronger languages. The latter is a direct consequence of the fact that reachability in trees is LOGSPACE-complete [8].

Much work has been dedicated to developing efficient algorithms for finding matches of tree patterns and tree homeomorphisms. Certain algorithms aim at processing the data tree as a stream (i.e., in a single scan) [5, 6, 13, 15, 9, 16, 2, 3, 17]. For this case a number of lower bound results have been obtained using mechanisms from communication complexity [2, 3, 14]. It is basically known that streaming algorithms for even simple tree patterns consume space proportional to the size of the data tree in the worst case. Table 1 lists algorithms for the tree homeomorphism matching problem together with bounds on their running time and space consumption. Here D is the data tree and Q is the tree pattern. We assume a random access machine model with unit cost for reading and writing integers. Some of the algorithms presented support generalizations of the tree homeomorphism problem but where a better bound is known for the tree homeomorphism problem, it is shown. Some of the streaming algorithms [3, 17] use a notion of candidate node sets cand_D which depends on the algorithm and which can be of size close to $|D|$ in the worst case. The algorithm of [3] makes the assumption of so-called non-recursive data trees, in which no two nodes such that one is a descendant of the other may have the same label. Finally, streaming algorithms such as [15] focus on being able to process SAX-events in constant time, at the cost of an exponential preprocessing step.

In this paper we study the tree homeomorphism (matching) problem. We establish a tight complexity characterization and develop an algorithm for the node-selection problem (shown at the bottom of Table 1) that is both time- and space efficient. In detail, the technical contributions of this paper are as follows.

– We first develop a top-down algorithm for the tree homeomorphism problem that is in LOGDCFL.[3]
– From this we develop a proof that the problem is LOGSPACE-complete, improving on the LOGCFL upper bound from [11].
– As our main result we develop a bottom-up LOGDCFL algorithm for computing all solutions of the tree homeomorphism problem which is both time and space efficient. This is a rather difficult algorithm and the correctness proof is involved. The algorithm runs in time $O(|D| \cdot |Q| \cdot \mathit{depth}(Q))$ and employs a stack of depth bounded by $\mathcal{O}(\mathit{depth}(D) \cdot \mathit{branch}(D))$.
  The algorithm may be of relevance in practical implementations. Indeed, in most Web or XML applications, the data tree is *much* larger than the tree pattern yet its depth is rather small. It can be observed that ours is the only algorithm in Table 1 — and to the best of our knowledge, in existence — that can guarantee a space bound that does not contain the size, but only depth and branching factor, of the data tree as a term. At the same time the algorithm admits a good time bound.
  Furthermore, the algorithm is of relevance in theory as well. It is a first step in classifying the complexity of positive Core XPath with child and descendant axis, which is probably the most widely used XPath fragment in practice. Its precise complexity, however, is unknown.
– In some applications (e.g., for certain XML data trees), a few nodes can have a very large number of children. Our algorithm can be made to run in space $O(\mathit{depth}(D) \cdot \log(\mathit{branch}(D)))$ with the same time bound if we assume the data tree to be in a ranked form that can be obtained by a LOGSPACE linear-time preprocessing algorithm. Given that ours is an offline algorithm it means little loss of generality to assume that data trees are kept in a database in this preprocessed form.

The paper presents these result basically in the order given here. Because of space limitations, some proofs had to be omitted.

2 Definitions

By $\mathbb{N}$ we denote the set of strictly positive integers. By Σ we denote a finite alphabet. The set of *unranked Σ-trees*, denoted by $\mathcal{T}_\Sigma$, is the smallest set of strings over Σ and the parenthesis symbols "(" and ")" which contains the

[3] For our purposes, it is enough to know that LOGDCFL is characterized by deterministic logspace bounded pushdown automata which run in polynomial time [18].

empty string and, for each $a \in \Sigma$ and $w \in (\mathcal{T}_\Sigma)^*$, contains $a(w)$. So, a tree is either ε (empty) or is of the form $a(T_1 \cdots T_n)$ where each T_i is a tree. In the tree $a(T_1 \cdots T_n)$, the subtrees $T_1, \ldots, T_n$ are attached to the root labeled a. When we write a tree as $a(T_1 \cdots T_n)$, we tacitly assume that every T_i is a non-empty tree. Moreover, we write a rather than $a()$. Notice that there is no a priori bound on the number of children of a node in a Σ-tree; such trees are therefore *unranked*. A *hedge* H is a finite sequence $T_1 \cdots T_n$ of trees. Hence, the set of *unranked Σ-hedges*, denoted by $\mathcal{H}_\Sigma$, equals $(\mathcal{T}_\Sigma)^*$. When we write a hedge as $T_1 \cdots T_n$, we tacitly assume that every T_i is a non-empty tree. In the sequel, whenever we say tree or hedge, we always mean Σ-tree or Σ-hedge, respectively. We will slightly abuse terminology and use the term "tree" to also refer to a hedge consisting of one tree, and we use the term "hedge" to also refer to the union of trees and hedges. We assume familiarity with terms such as *child*, *parent*, *descendant*, *ancestor*, *leaf*, *root*, *first child*, *last child*, *first sibling*, and *last sibling*.

For a hedge H, the *set of nodes* or *domain* of H, denoted by $\mathrm{Dom}(H)$, is the subset of $\mathbb{N}^*$ inductively defined as follows: *(i)* if $H = \varepsilon$, then $\mathrm{Dom}(H) = \emptyset$; *(ii)* if $H = a$, then $\mathrm{Dom}(H) = \{1\}$; *(iii)* if $H = a(T_1 \cdots T_n)$, where each $T_i \in \mathcal{T}_\Sigma - \{\varepsilon\}$, then $\mathrm{Dom}(H) = \{1\} \cup \bigcup_{i=1}^{n} \{1iu \mid 1u \in \mathrm{Dom}(T_i)\}$; and *(iv)* if $H = T_1 \cdots T_n$ with $n \geq 2$ and each $T_i \in \mathcal{T}_\Sigma - \{\varepsilon\}$, then $\mathrm{Dom}(H) = \{iu \mid 1u \in \mathrm{Dom}(T_i)\}$. The label of node u in the tree or hedge H, denoted by $\mathrm{lab}^H(u)$, is defined as follows: *(i)* if $H = a$ and $u = 1$, then $\mathrm{lab}^H(u) = a$; *(ii)* if $H = a(T_1 \cdots T_n)$ and $u = 1iv$ with $i \in \{1, \ldots, n\}$, then $\mathrm{lab}^H(u) = \mathrm{lab}^{T_i}(1v)$; and *(iii)* if $H = T_1 \cdots T_n$ with $n \geq 2$ and $u = iv$ with $i \in \{1, \ldots, n\}$, then $\mathrm{lab}^H(u) = \mathrm{lab}^{T_i}(1v)$.

By $|H|$, we denote the number of nodes in a hedge H. The *depth* of a node u in hedge H, denoted by $\mathrm{depth}^H(u)$, is 1 when $u \in \mathbb{N}$ and $1 + \mathrm{depth}(v)$ when $u = vi$ and $i \in \mathbb{N}$. The *height* of a node u in hedge H, denoted by $\mathrm{height}^H(u)$, is 1 when u is a leaf and $\max(\mathrm{height}^H(u1), \ldots, \mathrm{height}^H(uk)) + 1$ when u has $k > 0$ children. By $\mathrm{subtree}^H(u)$, we denote the subtree of H rooted at node u. In the remainder of the paper, we usually leave H implicit when H is clear from the context.

The Tree Homeomorphism Problem. A *tree pattern query* (with descendant edges) Q is an unranked tree over the alphabet $\Sigma \uplus \{*\}$. In the following, we use the terms *data tree* or *data hedge* to refer to ordinary Σ-trees and Σ-hedges. Given a data hedge H, a node $u \in \mathrm{Dom}(H)$, and a tree pattern query Q, we say that H *matches* Q *at node* u, denoted by $H \models^u Q$, if one of the following holds:

- $H = a$, $Q = a$ or $Q = *$, and $u = 1$;
- $H = a(T_1 \cdots T_n)$, $Q = a$ or $Q = *$, and $u = 1$;
- $H = a(T_1 \cdots T_n)$, $T_i \models^{1v} Q$, and $u = 1iv$, for some $i \in \{1, \ldots, n\}$;
- $H = T_1 \cdots T_n$, $T_i \models^{1v} Q$, and $u = iv$, for some $i \in \{1, \ldots, n\}$;
- $H = a(T_1 \cdots T_n)$, $Q = x(Q_1 \cdots Q_m)$, $u = 1$, $x \in \Sigma \uplus \{*\}$, $a \models x$, and, for every $k = 1, \ldots, m$, there exists an $i_k \in \{1, \ldots, n\}$, $u_k \in \mathrm{Dom}(T_{i_k})$, such that $T_{i_k} \models^{u_k} Q_k$.

Notice that the ordering of children in our tree pattern queries does not matter. This corresponds to the well known semantics of XPath queries with descendant

Algorithm 1 Tree pattern matching with descendant axes: Top-down algorithm Match

```
   Match (DNode $d$, QNode $q$)
2: if $d$ matches $q$ then
       return $\forall$ child $q_c$ of $q$ $\exists$ child $d_c$ of $d$: Match($d_c$,$q_c$)
4: else                          ▷ $q$ not matched yet, try $d$'s children
       return $\exists$ child $d_c$ of $d$: Match($d_c$,$q$)
6: end if
```

axes [7]. In the following, we abbreviate by $H \models Q$ that $H \models^u Q$ for some $u \in \text{Dom}(H)$. Alternatively, we say that H *matches* Q.

In this paper, we are interested in the following problems. Given a data tree T and a tree pattern query Q, the *tree homeomorphism problem* consists of deciding whether $T \models Q$. Furthermore, we are interested in *computing all answers* for the tree homeomorphism problem, that is, computing all nodes $u \in \text{Dom}(T)$ such that $T \models^u Q$. We refer to the latter problem as *tree homeomorphism matching.*

We assume that trees are stored on tape as a set of records; one for each node. Each record contains a pointer to its first child, last child, parent, previous sibling, and next sibling.

In the remainder of the paper, we assume a fixed data tree D and a fixed query tree Q for ease of presentation. We will refer to nodes of D and Q as *data nodes* and *query nodes*, respectively.

3 A Top-Down Algorithm

This section provides a simple top-down algorithm for the tree homeomorphism matching problem. The core of this top-down algorithm lies in a simple procedure that decides, given a data node d and a query node q, whether subtree(d) $\models$ subtree(q).

3.1 A Top-Down LOGDCFL Algorithm

Algorithm 1 describes the procedure Match to test whether subtree(d) $\models$ subtree(q). It is straightforward to prove that Match is indeed correct.

Lemma 1. Match *is correct. That is, given a data node d and a query node q,* Match *returns true iff subtree(d) $\models$ subtree(q).*

We can turn the procedure in Algorithm 1 into an algorithm Top-Down-Match for the tree homeomorphism matching problem as follows. First, we need a procedure Exact-Match that, given a data node d and query node q, decides whether subtree(d) $\models^1$ subtree(q). This is easy: Exact-Match only differs from Match in l.5, where it just returns false. Given a data node d and the root q_{root} of the query tree, Top-Down-Match now simply iterates over all the data nodes and returns every data node d for which Exact-Match(d, q_{root}) returns true. From this construction and from the correctness of Match, it is now immediate that Top-Down-Match is correct as well.

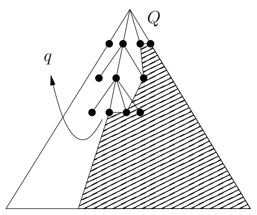


Fig. 1. Illustration of the remainder of q in Q.

Time and Space Complexity. It can be shown quite directly that the time complexities of Match and Exact-Match are in $\mathcal{O}(|\text{subtree}(d)| \cdot |\text{subtree}(q)|)$. As Top-Down-Match simply calls Exact-Match for every data node, we immediately have the following result.

Proposition 2. *The running time of* Top-Down-Match *is in* $\mathcal{O}(|D|^2 \cdot |Q|)$. *Moreover,* Top-Down-Match *makes* $\mathcal{O}(|D|^2 \cdot |Q|)$ *comparisons between a data node and a query node.*

It is immediate from our implementation of the algorithm that it can be executed by a deterministic logarithmic space bounded auxiliary pushdown automaton (see, e.g., [18]). Moreover, by Proposition 2, this auxiliary pushdown automaton runs in polynomial time. It follows from [18] that the tree homeomorphism matching problem is in LOGDCFL. As the maximum recursion depth of Algorithm 1 is $\mathcal{O}(\text{depth}(D))$, this renders the algorithm quite space-efficient, but the running time being quadratic in the size of the data tree, and the many unnecessary comparisons between query and data nodes are quite unsatisfactory. In the next section, we show how these issues can be resolved by turning to a bottom-up approach.

3.2 A LOGSPACE Procedure

While the top-down algorithm does not seem to be well-suited for efficiently computing *all* nodes u for which $D \models^u Q$, it is quite useful for *deciding* whether $D \models Q$, from a complexity theory point of view. Indeed, as we will exhibit, a modified version of Match can decide in LOGSPACE whether $D \models Q$. To this end, we assume the *left-to-right pre-order* ordering on nodes in trees and hedges in the remainder of this section. In particular, for every node u with k children in a hedge H, we have that $u < u1 < u2 < \cdots < uk$. For a node u, we denote by $u + 1$ the next node in the depth first, left-to-right traversal.

We argue how to transform Algorithm 1 into a LOGSPACE algorithm that decides whether $D \models Q$. Intuitively, the LOGSPACE algorithm processes the data and query trees in a top-down manner, just like Algorithm 1, and it processes the children of a node from left to right. The essential difference, however, lies in a backtracking procedure. When, for example, Algorithm 1 matches a leaf q of the query tree onto some data node d, then it uses the recursion stack to

Algorithm 2 LOGSPACE decision procedure: Top-down algorithm L-MATCH. We assume left-to-right preordering on trees.

```
L-MATCH (DNode $d$, QNode $q$)
if $d$ matches $q$, and both $d$ and $q$ have children then
    return L-MATCH ($d+1$,$q+1$)
else if $d$ does not match $q$ and $d$ has a child then
    return L-MATCH ($d+1$, $q$)
else if $d$ matches $q$ and $q$ is a leaf then
    if $q$ is maximal in $Q$ then return true   ▷ none of $q$'s ancestors has a right sib.
    else
        $d' \leftarrow$ BACKTRACK($d, q+1$)   ▷ node onto which $q+1$'s parent was matched
        return L-MATCH ($d'+1$, $q+1$)
    end if
else   ▷ $d$ is a leaf and ($d$ does not match $q$ or $q$ is not a leaf)
    if $d$ is maximal in $D$ then return false
    end if
    while $q$ has a parent do
        $d' \leftarrow$ BACKTRACK($d, q$)   ▷ node onto which $q$'s parent was matched
        if $d'$ is an ancestor of $d+1$ then return L-MATCH ($d+1$, $q$)
        else $q \leftarrow q$.parent
        end if
    end while
    return L-MATCH ($d+1$, $q$)
end if
```

discover the data node onto which q's parent was matched in the data tree and tries to match q's next sibling in some subtree of that data node. Instead of using this recursion stack, the LOGSPACE algorithm enters a subprocedure BACKTRACK(d, q) that *recomputes* d'. In particular, BACKTRACK(d, q) computes the highest possible node d'' on the path from D's root to d, such that the path from D's root to d'' matches the path from Q's root to q's parent. The crux of the algorithm is that this is *correct*, i.e., $d'' = d'$; and that BACKTRACK(d, q) can be performed using only logarithmic space on a Turing Machine. BACKTRACK(d, q) stores d and q on tape and goes to the roots of the query and data tree. It then matches the path to d with the path to q in a greedy manner. The crux of executing BACKTRACK(d, q) using logarithmic space lies in the following. If we arrive at a node u in D (resp., Q), we have to be able to determine the child of u that lies on the path to d (resp. q). To this end, we first store d (resp., q) in a temporary variable v. We now determine v's parent by scanning the input tape (i.e., we search a node with a child-pointer to v) and we overwrite v with v's parent. We continue following the parent relation in this fashion until we find u, at which point we return the value of v, which is a child of u.

We present the LOGSPACE algorithm in Algorithm 2. For ease of presentation, we have written the algorithm as a recursive procedure, but it can be implemented to only use logarithmic space. This can be seen by observing Algorithm 2: every recursive call to L-Match is a return-statement, so the algorithm does not change when the recursion stack is not used at all.

Let, for a query node q, the *remainder of q in Q* be the subhedge of Q consisting of the nodes $\{q' \mid q \leq q' \leq q_{\max}\}$, where $q_{\max}$ is the maximal query nodes w.r.t. the depth-first left-to-right ordering. We illustrate the remainder of q in Q in Figure 1. Given a data node d and a query node q, the algorithm first tries to match the remainder of q in Q consistently with what has already been matched in D (lines 2–11). If this fails, it either returns false (line 13), or enters a backtracking procedure (lines 15–21).

Lemma 3. *Algorithm 2 is correct. That is, given the roots d and q of a data D and query tree Q, Algorithm 2 decides whether $D \models Q$. Moreover, Algorithm 2 only uses logarithmic space.*

Theorem 4. *The tree homeomorphism problem is LOGSPACE-complete.*

4 The Bottom-up Algorithm

Although the previously presented top-down algorithms for tree homeomorphism matching are quite space-efficient, their time complexity is quite high and they involve quite a lot of recomputing of already obtained matchings, which is unsatisfactory. We therefore turn to a bottom-up matching approach which has the property that *no* obtained matchings between the data and query tree need to be recomputed, which leads to a better time complexity of the overall algorithm.

Before presenting the bottom-up algorithm for the tree homeomorphism matching problem in detail, we need to introduce several formal notions. As in the previous section, we first present an algorithm for the tree homeomorphism problem and then show how to change it into an algorithm for the tree homeomorphism matching problem.

In the present section, we assume the *left-to-right post-order* ordering on nodes in trees and hedges. In particular, for every node u with k children in a hedge H, we have that $u1 < u2 < \cdots < uk < u$. For a node u, we denote by $u+1$ the next node in the left-to-right postorder traversal. Hence, when we, e.g., use terminology such as "largest" and "smallest", we always assume the left-to-right post ordering. In this section, we also assume that XML documents are stored on tape in left-to-right postorder (or, alternatively, together with a left-to-right postorder index), which allows a random-access machine model to verify the left-to-right post-order ordering in constant time. For technical purposes, we also assume two dummy nodes in every tree and hedge: nil and ∞. The node nil is such that nil$+1$ is the smallest node in the hedge, and the node ∞ is defined as the successor of the largest node of the hedge. Given two nodes $h_{\text{from}} \leq h_{\text{until}}$ in a hedge H, we denote by the interval $[h_{\text{from}}, h_{\text{until}}]$ the subhedge of H consisting only of the nodes $\{v \mid h_{\text{from}} \leq v \leq h_{\text{until}}\}$. The notion of such an interval in a tree is illustrated in Figure 2(a). Here, the interval $[h_{\text{from}}, h_{\text{until}}]$ is the striped area in the tree. Given a hedge H and a node $h \in \text{Dom}(H)$, we denote by $\text{subhedge}^H(h)$ the subhedge $[h_{\text{from}}, h]$, where h_{from} is the smallest descendant of h's leftmost

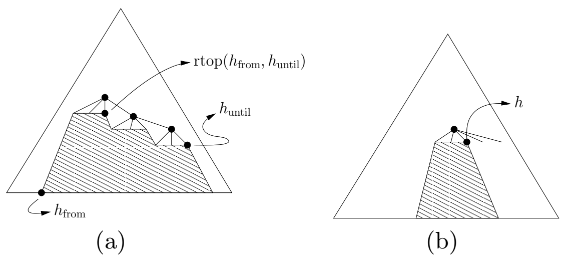


Fig. 2. Illustration of a hedge interval and RTOP (left) and of subhedge$^H(h)$ (right).

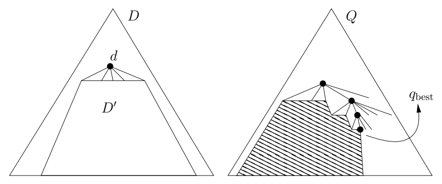
sibling according to the left-to-right postorder ordering. We illustrate this notion in Figure 2(b).

When H is a data hedge or a tree pattern query, we refer to $[h_{\text{from}}, h_{\text{until}}]$ as a data or query hedge interval, respectively. We extend the semantics of tree pattern matching to hedges as follows. Let $Q_1 \cdots Q_n$ be a query hedge interval $[q_{\text{from}}, q_{\text{until}}]$ and $D_1 \cdots D_m$ be a data hedge interval $[d_{\text{from}}, d_{\text{until}}]$. We say that $[d_{\text{from}}, d_{\text{until}}]$ matches $[q_{\text{from}}, q_{\text{until}}]$, denoted by $[d_{\text{from}}, d_{\text{until}}] \models [q_{\text{from}}, q_{\text{until}}]$, if, for every Q_i, $i = 1, \ldots, n$, there exists a D_j, $j = 1, \ldots, m$, such that $D_j \models Q_i$.

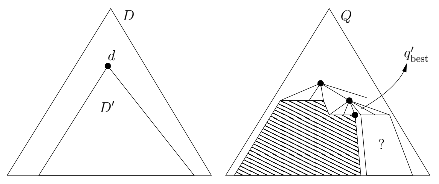
Before presenting the intuition about the bottom-up tree homeomorphism algorithm, we describe an auxiliary procedure RTOP, which, given two nodes h_{from} and h_{until}, returns the rightmost node among the topmost nodes in the interval $[h_{\text{from}}, h_{\text{until}}]$. More formally, RTOP$(h_{\text{from}}, h_{\text{until}})$ is the node u such that depth(u) is minimal and u is larger than every other node v in $[h_{\text{from}}, h_{\text{until}}]$ with depth(u) = depth(v). This notion is illustrated in Figure 2(a). Furthermore, in order to simplify the presentation of the algorithm, we define RTOP$(h_{\text{from}}, h_{\text{until}}) = \infty$ if $h_{\text{from}} > h_{\text{until}}$. Notice that RTOP can easily be computed in time linear in the depth of the tree and in logarithmic space by traversing the path from h_{until} to the query root and comparing the previous siblings of nodes on the path with h_{from} w.r.t. the left-to-right post-ordering. Indeed, assume that $h_{\text{from}} \leq h_{\text{until}}$. Let u be the highest ancestor of h_{until} that has a previous sibling s such that $s \geq h_{\text{from}}$. If no such u exists, then rtop$(h_{\text{from}}, h_{\text{until}})$ is h_{until}. Otherwise, rtop$(h_{\text{from}}, h_{\text{until}})$ is s.

We first present an algorithm for *deciding* whether $D \models Q$ and show later how it can be extended to an algorithm for the tree homeomorphism matching problem. The main procedure of our algorithm is called TMATCH. Given a data node d and query nodes q_{from} and q_{until}, TMATCH returns the largest query node q in the interval $[q_{\text{from}}, q_{\text{until}}]$ such that subtree$^D(d)$ matches $[q_{\text{from}}, q]$ if q exists; and $q_{\text{from}} - 1$ otherwise. Hence, if d is the root of D, and q_{from} and q_{until} are the leftmost leaf and the root of Q, respectively, then $D \models Q$ if and only if TMATCH returns q_{until}.

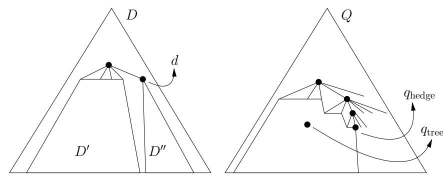
TMATCH uses an auxiliary procedure called HMATCH, which, given a data node d and query nodes q_{from} and q_{until}, returns the largest node q in the interval $[q_{\text{from}}, q_{\text{until}}]$ such that subhedge$^D(d)$ matches $[q_{\text{from}}, q]$ if q exists; and $q_{\text{from}} - 1$ otherwise.



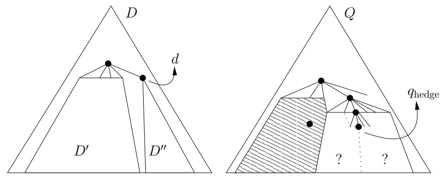
(a) Operation of TMatch: recursive call of HMatch.



(b) Operation of TMatch: recursive call of TMatch.



(c) Operation of HMatch: first recursive calls of TMatch and HMatch.



(d) Operation of HMatch: a subsequent recursive call of TMatch, trying to improve q_{tree}.

Fig. 3. Illustrations of the tree homeomorphism algorithm.

We start by explaining the operation of TMatch, which is presented in Algorithm 3. Given a data node d and query nodes q_{from} and q_{until}, TMatch first starts by recursively calling HMatch with the same query nodes for the subhedge D' of D defined by d's last child, yielding result q_{best} (see Figure 3(a)). In the remainder of TMatch, we essentially want to test how q_{best} can be improved when we also consider the node d in addition to D'. One particular interesting case is when q_{best} is a last sibling and its parent has the same label as d. In this case, we can at least improve our best query node to q_{best}'s parent which we call here q'_{best}. Furthermore, it is possible that q'_{best} is not yet the best query node we can obtain. In particular, we still need to test which part of the hedge defined by $[q'_{\text{best}}+1, q'_{\text{best}}.\text{lastSibling}]$ can be matched in the subtree below d (see Figure 3(b)). The largest node that is obtained in this manner is the node that TMatch should return.

We now explain the operation of HMatch, which is presented in Algorithm 4. Essentially, given d, q_{from}, and q_{until}, HMatch starts by recursively calling itself with the same query nodes on the hedge defined by the previous sibling of d (i.e., D' in Figure 3(c)), yielding q_{hedge}, and by calling TMatch with the same query nodes on the subtree under d itself (D'' in Figure 3(c)), yielding q_{tree}. The remainder of HMatch consists of iteratively improving q_{tree} and q_{hedge}. That is, while it is possible that D' and D'' yield small values of q_{tree} and q_{hedge}, their concatenation can give rise to a much larger part of the query that can be matched. Essentially, this is due to the fact that the matching of tree pattern queries is *unordered*. For example, it can occur that we need to match a certain first sibling in D', a second one in D'', a third one again in D' and so

Algorithm 3 Tree pattern matching: function TMatch.

```
TMatch (DNode $d$, QNode $q_{\text{from}}$, QNode $q_{\text{until}}$)
if $d$ is a leaf then $q_{\text{best}} \leftarrow q_{\text{from}} - 1$
else $q_{\text{best}} \leftarrow$ HMatch($d$.lastChild, $q_{\text{from}}$, $q_{\text{until}}$)
end if
if $q_{\text{best}} + 1 \leq q_{\text{until}}$ and $d$ matches $q_{\text{best}} + 1$ then
    $q_{\text{best}} \leftarrow q_{\text{best}} + 1$
    if $q_{\text{best}} + 1 \leq q_{\text{best}}$.lastSib then
        return TMatch($d$, $q_{\text{best}} + 1$, $q_{\text{best}}$.lastSib)
    else return $q_{\text{best}}$
    end if
else return $q_{\text{best}}$
end if
```

on. Hence, the procedure HMatch alternates between finding best matches in D' and D'' until it reaches a fixpoint.

However, we need to take care in how this fixpoint is computed. One possible case is illustrated in Figure 3(d). This particular case builds further on the situation in Figure 3(c). Here, we try to improve q_{tree} by starting the TMatch procedure again for the node d, but now only with the part of the query marked with question marks. The case where q_{tree} is larger than q_{hedge} is dual and not illustrated here.

Example 5. Figure 4(a) and 4(b) illustrate an example for the bottom up algorithm. For brevity, we denote TMatch and HMatch with TM and HM, respectively. The first calls of TM and HM demonstrate the basic recursive structure of our algorithm: TM on a node d calls HM on the rightmost child of d. HM on a node d returns TM of d if that node is a first sibling; or performs a divide-and-conquer technique by calling HM on the left sibling of d and TM on d itself (as in the function call HM(d_4, q_1, q_5)). Further recursive calls to TM or HM are then needed to maximize the part of the query that can be matched.

The simplest function call in the example that performs such further recursive calls is the call HM(d_2, q_1, q_5), which starts by computing $q_{\text{hedge}} = \text{HM}(d_1, q_1, q_5)$ and $q_{\text{tree}} = \text{TM}(d_2, q_1, q_5)$. As can be seen in Figure 4(b), $q_{\text{hedge}} = \text{nil}$. The call TM$(d_2, q_1, q_5)$ is more successful, because d_2 and q_1 are both labeled with a. In general, it might be possible that q_2 and further nodes can be matched in subtree(d_2). The function call TM(d_2, q_2, q_4) checks that possibility. (For sure, q_1 and q_5 cannot both be matched on d_2, which is why we restrict the query tree interval by q_4.) But q_2 is not labeled with a so the return value of the two TM calls is q_1. After this initial phase, HM(d_2, q_1, q_5) tries to improve q_{tree} and q_{hedge} iteratively. It calls HM(d_1, q_2, q_4) and improves q_{hedge} to be q_2, because q_2 and d_1 are both labeled with b. Further improvements fail as there is no c-labeled node in the subhedge of d_2.

A similar iterative improvement is illustrated by HM(d_3, q_1, q_5). Observe that we try to improve q_{tree} here and call TM(d_4, q_2, q_4) and TM(d_4, q_3, q_3). Only the latter call yields an improvement. But we cannot omit the former one: if

Algorithm 4 Tree pattern matching: function HMatch.

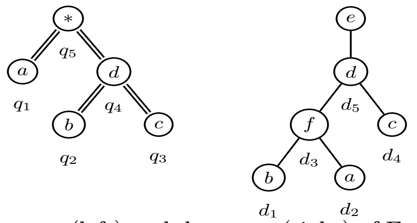
```
    HMatch (DNode $d$, QNode $q_{\text{from}}$, QNode $q_{\text{until}}$)
14: if $d$ is a first sibling then return TMatch($d$, $q_{\text{from}}$, $q_{\text{until}}$)
    else
16:   $q_{\text{hedge}} \leftarrow$ HMatch($d$.prevSib, $q_{\text{from}}$, $q_{\text{until}}$)
      $q_{\text{tree}} \leftarrow$ TMatch($d$, $q_{\text{from}}$, $q_{\text{until}}$)
18:   loop
        if $q_{\text{hedge}} = q_{\text{tree}}$ then return $q_{\text{hedge}}$
20:     else if $q_{\text{tree}} < q_{\text{hedge}}$ then
          rtop $\leftarrow$ RTop($q_{\text{tree}} + 1, q_{\text{hedge}}$)
22:       while rtop $< \infty$ and $q_{\text{hedge}} <$ rtop.lastSib do
            $q_{\text{tree}} \leftarrow$ TMatch($d$, rtop+1, rtop.lastSib)
24:         rtop $\leftarrow$ RTop($q_{\text{tree}} + 1, q_{\text{hedge}}$)
          end while
26:       if $q_{\text{tree}} \leq q_{\text{hedge}}$ then return $q_{\text{hedge}}$
          end if
28:     else
          rtop $\leftarrow$ RTop($q_{\text{hedge}} + 1, q_{\text{tree}}$)
30:       while rtop $< \infty$ and $q_{\text{tree}} <$ rtop.lastSib do
            $q_{\text{hedge}} \leftarrow$ HMatch($d$.prevSib, rtop $+ 1$, rtop.lastSib)
32:         rtop $\leftarrow$ RTop($q_{\text{hedge}} + 1, q_{\text{tree}}$)
          end while
34:       if $q_{\text{hedge}} \leq q_{\text{tree}}$ then return $q_{\text{tree}}$
          end if
36:     end if
      end loop
38: end if
```

subtree(d_4) would match subtree(q_4), then the former call would yield q_4 and the latter call would yield q_3. As we want our algorithm to return the largest query node such that the interval ending with it can be matched the result of the former call would have been the relevant one in that case.

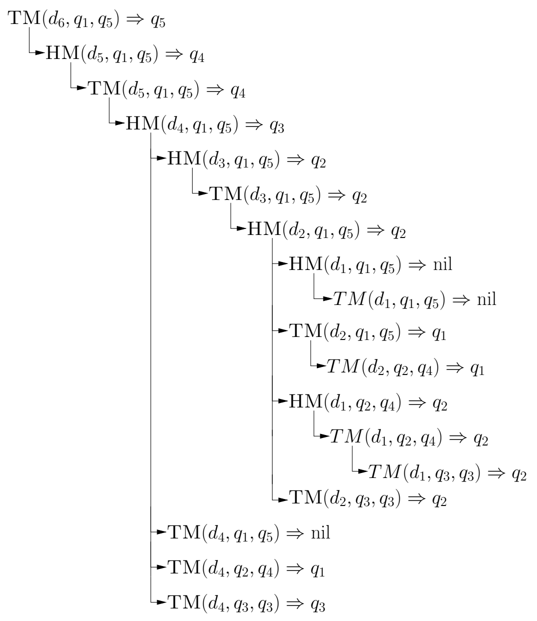
Correctness. The main technical difficulty of the paper is proving that TMatch is correct. Due to space limitations, the proof has been omitted.

Lemma 6. *Let D be a data tree and let Q be a query tree.* TMatch *is correct, that is, given the root node d of D, the smallest and largest node q_{from} and q_{until} of Q, respectively,* TMatch *returns q_{until} iff $D \models Q$.*

We now argue how TMatch can be modified to a procedure TMatch-All, that computes *all* data nodes u such that $D \models^u Q$. In order to compute *all* the matches, we add a test to l.9 of TMatch. That is, before returning q_{best}, we test whether q_{best} is the root of Q, and we output d if it is. Now we return $q_{\text{best}} - 1$, as if the query root was not matched. Furthermore, TMatch-All recursively calls TMatch-All and HMatch-All instead of TMatch and HMatch. Here HMatch-All is the same as HMatch, except that it recursively calls TMatch-All and HMatch-All instead of HMatch and TMatch.



(a) Query tree (left) and data tree (right) of Example 5.



- $\mathrm{TM}(d_6, q_1, q_5) \Rightarrow q_5$
  - $\mathrm{HM}(d_5, q_1, q_5) \Rightarrow q_4$
    - $\mathrm{TM}(d_5, q_1, q_5) \Rightarrow q_4$
      - $\mathrm{HM}(d_4, q_1, q_5) \Rightarrow q_3$
        - $\mathrm{HM}(d_3, q_1, q_5) \Rightarrow q_2$
          - $\mathrm{TM}(d_3, q_1, q_5) \Rightarrow q_2$
            - $\mathrm{HM}(d_2, q_1, q_5) \Rightarrow q_2$
              - $\mathrm{HM}(d_1, q_1, q_5) \Rightarrow \mathrm{nil}$
                - $TM(d_1, q_1, q_5) \Rightarrow \mathrm{nil}$
              - $\mathrm{TM}(d_2, q_1, q_5) \Rightarrow q_1$
                - $TM(d_2, q_2, q_4) \Rightarrow q_1$
              - $\mathrm{HM}(d_1, q_2, q_4) \Rightarrow q_2$
                - $TM(d_1, q_2, q_4) \Rightarrow q_2$
                  - $TM(d_1, q_3, q_3) \Rightarrow q_2$
              - $\mathrm{TM}(d_2, q_3, q_3) \Rightarrow q_2$
        - $\mathrm{TM}(d_4, q_1, q_5) \Rightarrow \mathrm{nil}$
        - $\mathrm{TM}(d_4, q_2, q_4) \Rightarrow q_1$
        - $\mathrm{TM}(d_4, q_3, q_3) \Rightarrow q_3$

(b) Function calls of HMatch (HM) and TMatch (TM) of Example 5.

Fig. 4. Illustrations for Example 5.

The following theorem can be proved:

Theorem 7. *Let d be the root node of D and let q_{from} be the smallest and q_{root} be the largest node of Q, respectively.* TMatch-All *is correct, that is,* TMatch-All(d, q_{from}, q_{until}) *outputs the data nodes u such that $D \models^u Q$.*

Proof (Sketch). It follows directly from our additional test and the correctness of TMatch that $D \models^u Q$ for all the nodes u that TMatch-All outputs.

It remains to prove that, if $D \models^u Q$, then TMatch-All outputs u. Towards a contradiction, assume that there is an u such that $D \models^u Q$, but u was not reported by TMatch-All. By an easy induction it can be shown that for every data node d_0 in D there is a call TMatch-All for d_0's subtree and Q. In particular, there was a call TMatch-All$(u, q_{\mathrm{from}}, q_{\mathrm{root}})$. Since this call did not output

u, it follows that u must have children and that HMatch-All(u.lastChild, q_{from}, q_{root}) $< q_{\text{root}} - 1$, (because otherwise q_{root} and u would have been compared and u would have been written to the output). In general, we have that HMatch-All(d, q_1, q_2) $= \min$ (HMatch(d, q_1, q_2), $q_{\text{root}} - 1$).
It follows that HMatch-All(u.lastChild, $q_{\text{from}}, q_{\text{root}}$) = HMatch($u$.lastChild, $q_{\text{from}}, q_{\text{root}}$).

If we now call TMatch($u, q_{\text{from}}, q_{\text{root}}$), it calls HMatch($u$.lastChild, q_{from}, q_{root}), which yields again a value less than $q_{\text{root}} - 1$. Therefore, the return value of TMatch($u, q_{\text{from}}, q_{\text{root}}$) is less than q_{root}. But we assumed that subtree(u) $\models Q$, which contradicts the correctness of TMatch proved in Lemma 6. □

Time and Space Complexity. First, we need to show that our algorithm determines in PTIME whether $D \models Q$. Notice that the naïve manner of computing the running time of TMatch gives rise to only an exponential upper bound. Indeed, define *(i)* $T(N)$ as the running time of TMatch on d, q_{from}, and q_{until}, where subtree(d) and $[q_{\text{from}}, q_{\text{until}}]$ have N nodes in total, and *(ii)* $H(N)$ as the running time of HMatch on d, q_{from}, and q_{until}, where subhedge(d) and $[q_{\text{from}}, q_{\text{until}}]$ have N nodes in total. Then, we have that $T(2) \leq p(N)$ for a polynomial p, $T(N) \leq p(N) + H(N-1) + T(N-1)$, and $H(N) \leq T(N) + X(N)$, where $X(N) \geq 0$. Hence, $T(N) \leq 2^{N-1}$, which is obviously not sufficient.

We therefore employ a slightly more sophisticated approach in the following Lemma.

Lemma 8. *Given the root node of a data tree D, and the smallest and largest query nodes and of a query tree Q, respectively,* TMatch *runs in time $\mathcal{O}(|D| \cdot |Q| \cdot depth(Q))$. Moreover,* TMatch *makes $\mathcal{O}(|D| \cdot |Q|)$ comparisons between a data node and a query node.*

The depth(Q) factor in the complexity of TMatch is due to the calls to rtop in HMatch, and the computation of the successors of query nodes.

Theorem 9. TMatch-All(D, Q) *runs in time $\mathcal{O}(|D| \cdot |Q| \cdot depth(Q))$. Moreover,* TMatch-All *makes $\mathcal{O}(|D| \cdot |Q|)$ comparisons between a data node and a query node.*

Currently, the maximum recursion depth of TMatch-All is $\mathcal{O}(\text{depth}(D) \times \text{branch}(D))$, where branch($D$) is the maximum number of children a node in D has. We have the branch(D) factor because HMatch($d, q_{\text{from}}, q_{\text{until}}$) calls HMatch($d$.prevSib, $q_{\text{from}}, q_{\text{until}}$). However, this bound can be improved using a simple preprocessing step: we can turn D into a binary tree D_{bin} by inserting intermediate levels of special nodes between each data node and its children. By doing so, D only grows linearly in size and the depth only grows by a factor of log(branch(D)).

As Q only uses descendant axes, we have that $D \models^u Q$ iff $D_{\text{bin}} \models^u Q$.[4] When this preprocessing step is carried out, our algorithm still has $\mathcal{O}(|D||Q|\text{depth}(Q))$

[4] Under the assumption that the new dummy nodes do not match $*$, which can be trivially incorporated in the algorithm.

time complexity, but the recursion/stack depth is improved to $\mathcal{O}(\text{depth}(D) \cdot \log(\text{branch}(D)))$.

References

1. M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proc. VLDB*, pages 53–64, 2000.
2. Z. Bar-Yossef, M. Fontoura, and V. Josifovski. On the memory requirements of XPath evaluation over XML streams. In *Proc. PODS*, pages 177–188, 2004.
3. Z. Bar-Yossef, M. Fontoura, and V. Josifovski. Buffering in query evaluation over XML streams. In *Proc. PODS*, 2005.
4. N. Bruno, D. Srivastava, and N. Koudas. Holistic twig joins: Optimal XML pattern matching. In *Proc. SIGMOD*, pages 310–321, 2002.
5. C. Y. Chan, W. Fan, P. Felber, M. N. Garofalakis, and R. Rastogi. Tree pattern aggregation for scalable XML data dissemination. In *Proc. VLDB*, pages 826–837, 2002.
6. C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proc. ICDE*, pages 235–244, 2000.
7. J. Clark and S. DeRose. XML Path Language (XPath). Technical report, World Wide Web Consortium, November 1999. http://www.w3.org/TR/xpath.
8. S. A. Cook and P. McKenzie. Problems complete for deterministic logarithmic space. *J. Algorithms*, 8:385–394, 1987.
9. Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.
10. G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.
11. G. Gottlob, C. Koch, R. Pichler, and L. Segoufin. The complexity of XPath query evaluation and XML typing. *J. ACM*, 52(2):284–335, 2005.
12. G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. *J. ACM*, 48(1):431–498, 2001.
13. T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.
14. M. Grohe, C. Koch, and N. Schweikardt. Tight lower bounds for query processing on streaming and external memory data. In *Proc. ICALP 2005*, 2005.
15. A. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *Proc. SIGMOD*, pages 419–430, 2003.
16. D. Olteanu, T. Furche, and F. Bry. An evaluation of regular path expressions with qualifiers against XML streams. In *Proc. BNCOD 2004*, pages 31–44, 2004.
17. P. Ramanan. Evaluating an XPath query on a streaming XML document. In *Proc. COMAD 2005*, pages 41–52, 2005.
18. I. H. Sudborough. Time and tape bounded auxiliary pushdown automata. In *Proc. MFCS 1977*, pages 493–503. Springer Verlag, 1977.
19. M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. VLDB*, pages 82–94, 1981.